Projet

1 Objectif

Le projet de Complément de programmation a comme objectifs principaux de :

1. approfondir les notions d'algorithmique et de programmation vues cette année ;
2. pratiquer le travail en groupe, qui est un mode de travail courant dans les métiers de l'informatique (et beaucoup d'autres) ;
3. vous donner une première expérience du développement professionnel (développement itératif, compilation, conception, documentation, test).

La mise en œuvre se fait au travers de la réalisation d'un logiciel. Il s'agit d'un jeu inspiré directement de la bataille navale.

Le projet est réalisé par groupe de 4 personnes (éventuellement 3 selon l'effectif du TD), mais la note est individuelle.

A la fin de chaque itération, vous déposerez sur UPdago différents fichiers dans une archive zip :

— un fichier pdf comprenant : la conception, le "qui fait quoi" et le rapport des tests manuels (tests unitaires et d'intégration des fonctions graphiques et tests fonctionnels de fin d'itération)
— le fichier des sources `battleship.ml`,
— le fichier des tests automatisés `battleship_test.ml`,
— la documentation générée dans un répertoire dédié.

Attention : toutes les séances de projet sont notées. Les éventuelles absences sont à justifier avec la même rigueur que tout contrôle.

2 Travail en groupe

La réalisation de chaque itération se fera en 3 phases :

1. une étape de conception, où vous travaillez en groupe dont le but est de lister les types et fonctions à coder (ce qui nécessite de prévoir les différentes étapes des algorithmes), puis de définir les types et les signatures des fonctions, et les documenter,
2. une étape de réalisation, dont le but est de coder les fonctions sources et tests,
3. une étape d'intégration, de test et de mise au point.

La première étape est une étape de travail collectif. Pour être efficace, chacun doit arriver en ayant (re)lu le sujet et la description de l'itération. Chacun explique ce qu'il a compris. Si des différences de compréhension persistent à l'issue des échanges il convient d'interroger votre enseignant afin de bien identifier les besoins.

Une fois l'analyse des besoins réalisées, la conception peut commencer. Là aussi chacun doit s'exprimer, proposer et critiquer les propositions des autres. Pour pouvoir se partager le travail de développement à l'issue de la conception, il est nécessaire d'avoir suffisamment découpé le code en fonctions et sous-fonctions et d'avoir spécifier chacune de manière suffisamment précise. Les types doivent être complètement définis, pour être partagés entre différents développeurs.

A la fin de cette phase de conception le travail de développement est répartit entre les différents membres du groupe et inscrit dans le "qui fait quoi".

Pendant la deuxième phase chacun doit coder les parties qui lui sont affectées sans attendre le dernier jour.

Il peut arriver (surtout quand on débute), que la conception soit incomplète ou inadaptée. Auquel cas il faut alerter l'équipe au plus vite et revenir sur l'étape de conception, puis reprendre la réalisation sur de bonnes bases.

Chacun débug ses fonctions à l'aide de tests unitaires.

L'intégration ne doit pas se faire à la dernière minute, car elle prend du temps. Les différentes fonctions sources sont rassemblées dans un même fichier sources et les fonctions de test dans un même fichier de tests. Les tests sont exécutés.

En cas d'erreur, chaque développeur corrige son code. Avant de reprendre l'intégration et le test.

Il peut arriver que certaines erreurs ne soient pas dues au codage, mais à une conception erronée. Dans ce cas il convient de revenir à la phase de conception, avant de corriger le code, puis de reprendre l'intégration et le test.

Une fois les tests d'intégration en succès, le projet est compilé puis les tests fonctionnels réalisés. La documentation de l'ensemble du projet (sources et tests) est générée. Puis l'ensemble des fichiers sont déposés sur UPdago dans une archive zip.

Pour organiser le travail en équipe, activez votre compte Office 360 avec votre identifiant universitaire et créez une équipe Teams qui regroupe les membres de l'équipe. Vous pourrez y déposer l'ensemble de vos fichiers individuels et d'équipe, utiliser les outils de communication, etc.

En début de chaque itération nommez un **maître de mêlée** qui anime le travail d'équipe. Pendant la première phase, il anime les débats et s'assure que tout le monde s'exprime. Pendant la seconde phase, il s'assure que chaque développeur respecte les délais. Il organise la communication dans l'équipe et favorise l'entraide entre ses membres. En cas de problème il alerte le groupe et organise une réunion (en présentiel ou en visio) si un retour sur la conception est nécessaire. C'est lui qui note la répartition des responsabilités et des tâches dans le "qui fait quoi".

Nommez un **responsable de produit**. Il est responsable de l'analyse des besoins et de la conception. Il note les idées du groupe dans un fichier partagé. Il s'assure que l'ensemble des membres de l'équipe aient bien compris les différentes fonctions à réaliser.

Nommez un **responsable de l'intégration** qui rassemble le code des participants, exécute les tests, informe l'équipe des résultats, constitue l'archive de l'itération et la dépose sur UPdago. Ce n'est pas à lui de corriger les erreurs, mais à l'ensemble de l'équipe.

Les membres de l'équipe doivent participer à tous les types d'activité du projet : responsabilités, code source, test, documentation, relecture de code, etc. Il est donc important que les responsabilités et le type d'activité de chacun varie à chaque itération. A titre indicatif, le barème est d'environ 50% sur le code et 50% sur le génie logiciel (test, documentation, travail d'équipe...). Autrement dit, un étudiant qui ne fera que du codage sera donc noté sur 10 et non sur 20.

3 Environnement graphique

Pour avoir un jeu inter-actif, nous allons utiliser les bibliothèques OCaml `Graphics` et `Unix`. Ces dernières ne sont pas disponibles sous Jupyter et nécessitent une installation locale d'OCaml. Les documentations de ces bibliothèques sont disponibles en ligne et vous pouvez les installer sur votre machine personnelle à l'aide du gestionnaire de paquet opam :
`https://ocaml.org/manual/4.03/libref/Graphics.html`
`https://ocaml.org/manual/5.2/api/Unix.html`

Contrairement aux modules OCaml vus jusque ici, comme `String` ou `Random`, qui sont inclus dans la version de base d'OCaml et donc disponibles dès que l'on lance OCaml, les bibliothèques

`Graphics` et `Unix` doivent êtres chargées avant d'être utilisées. Sur les machines de TP, ces bibliothèques sont automatiquement chargées au lancement de l'interpréteur OCaml. Mais sur vos machines personnelles ce ne sera pas le cas. Pour les versions récentes d'OCaml (à partir de la version 4.09), le chargement se fait comme suit :

```
#use "topfind" ;;
#require "graphics" ;;
#require "unix" ;;
```

Pour les anciennes versions d'OCaml, le chargement se fait avec les commandes suivantes :

```
#load "graphics.cma";;
#load "unix.cma";;
```

Pour faciliter l'utilisation de ces 2 bibliothèques, téléchargez le fichier `CPgraphics.ml` et chargez le ensuite avec la commande :

```
#mod_use "CPgraphics.ml";;
```

Pour plus de facilité, mettez l'ensemble de ces commandes dans un fichier `inter.ml` que vous pourrez charger avec la commande `#use "inter.ml"` à chaque début de séance de travail.

Le fichier `CPgraphics.ml` fournit les fonctions suivantes :
`wait (n : int) : unit` permet de mettre le programme en pause pendant `n` secondes.
`open_graph (dx, dy : int * int) : unit` ouvre une fenêtre graphique de `dx` pixels par `dy` pixels. Le pixel `(0, 0)` est en bas à gauche, le pixel en haut à droite est à la position `(dx - 1, dy - 1)`.
`set_window_title (s : string) : unit` modifie le titre de la fenêtre graphique à `s`.
`close_graph () : unit` ferme la fenêtre graphique.
`clear_graph () : unit` nettoie la fenêtre graphique.
`resize_window (dx, dy : int * int) : unit` redimensionne la fenêtre graphique à la taille `dx` par `dy`.
`plot(x, y : int * int) : unit` colore le pixel `(x, y)` de la couleur courante.
`moveto (x, y : int * int) : unit` positionne le point courant sur le pixel `(x, y)`.
`current_point () : int * int` retourne la position du point courant.
`lineto (x, y : int * int) : unit` trace une ligne de la couleur courante du point courant jusqu'au pixel `(x, y)` et met à jour le point courant au pixel `(x, y)`.
`draw_circle (x, y, r : int * int * int) : unit` trace un cercle de centre `(x, y)` et de rayon `r`. La position courante est inchangée. L'exception `Invalid_argument` est levée si `r` est négatif.
`draw_ellipse (x, y, rx, ry : int * int * int * int) : unit` trace une ellipse de centre `(x, y)` et de rayons `rx` et `ry`. La position courante est inchangée. L'exception `Invalid_argument` est levée si `rx` ou `ry` sont négatifs.
`draw_rect (x, y, w, h : int * int * int * int) : unit` trace un rectangle dont le point en bas à gauche est à la position `(x, y)`, de largeur `w` et de hauteur `h`. La position courante est inchangée. L'exception `Invalid_argument` est levée si `w` ou `h` sont négatifs.
`fill_circle (x, y, r : int * int * int) : unit` rempli le cercle de centre `(x, y)` et de rayon `r`. La position courante est inchangée. L'exception `Invalid_argument` est levée si `r` est négatif.
`fill_ellipse (x, y, rx, ry : int * int * int * int) : unit` rempli l'ellipse de centre `(x, y)` et de rayons `rx` et `ry`. La position courante est inchangée. L'exception `Invalid_argument` est levée si `rx` ou `ry` sont négatifs.
`fill_rect (x, y, w, h : int * int * int * int) : unit` rempli le rectangle dont le point en bas à gauche est à la position `(x, y)`, de largeur `w` et de hauteur `h`. La position courante est inchangée. L'exception `Invalid_argument` est levée si `w` ou `h` sont négatifs.
`draw_char (c : char) : unit` trace la lettre `c` à partir de la position courante en bas à gauche de la lettre. Après exécution, la position courante est en bas à droite de la lettre.

`draw_string (s : string) : unit` trace le texte `s` à partir de la position courante en bas à gauche du texte. Après exécution, la position courante est en bas à droite du texte.
`set_text_size (n : int) : unit` mets à jour la taille courante d'écriture des textes.

Le type `t_color` défini les couleurs.
Les constantes `black`, `blue`, `orange`, `red`, `green`, `white`, `yellow`, `cyan`, `magenta` et `grey` sont des couleurs prédéfinies.
`color_of_rgb (r, g, b : int * int * int) : t_color` permet de définir la couleur dont la composante rouge est `r`, la composante verte est `g` et la composante bleue est `b`. `r`, `g` et `b` doivent être compris entre 0 et 255 et sont conformes à la représentation habituelle des couleurs RGB.
`set_color (color : t_color) : unit` met à jour la couleur courante. La couleur de départ est `black`.

`key_pressed () : bool` retourne `true` si une touche du clavier est enfoncée et `false` sinon. Elle permet d'éviter le blocage de la fonction `read_key`.
`read_key () : char` retourne le premier caractère du tampon de saisie. Si le tampon est vide, la fonction attend qu'une touche du clavier soit enfoncée et retourne le caractère correspondant.
`mouse_pos () : int * int` retourne la position courante de la souris. Si la souris est en dehors de la fenêtre graphique, la position retournée sera aussi en dehors de la fenêtre graphique.
`button_down () : bool` retourne `true` si un bouton de la souris est pressé et `false` sinon.
`wait_button_down () : int * int` attend le prochain clic de souris et retourne sa position.

4 Compilation

Nous avons vu en cours que la compilation se fait en deux étapes :

1. la compilation proprement dit, qui produit le bytcode d'un fichier OCaml,
2. l'édition de lien, qui rassemble les bytcode de tous les fichiers pour produire l'exécutable.

Lorsqu'un fichier sources n'utilise que les bibliothèques de base qui sont automatiquement chargées (comme `String`, `Array`, `List`...), nous n'avons pas besoin d'indiquer les dépendances. Par exemple on peut compiler la le fichier `AP1array` avec la commande :

```
ocamlc -c AP1array.ml
```

Mais lorsque un fichier sources utilise d'autres bibliothèques, il faut expliciter les dépendances. Par exemple, pour compiler `CPgraphics.ml` qui utilise les bibliothèques `Unix` et `Graphics`, il convient d'utiliser l'option `-I` afin d'indiquer où trouver les fichiers `unix.cma` et `graphics.cma` (qui ne sont pas dans le répertoire courant). Pour la bibliothèque `Unix` qui est récente, on peut utiliser l'option `-I +unix` qui ajoute le chemin de la bibliothèque aux chemins pré-définis pour OCaml. Mais pour la bibliothèque `Grapics` qui est un peu plus ancienne et qui n'est pas installée au même endroit, il convient d'indiquer le chemin exacte où la trouver. Sur les machines de TP elle se trouve dans `/opt/opam/default/lib/graphics`, mais sur ma machine elle est dans `~/.opam/default/lib/graphics`, ce qui probablement aussi le cas sur votre machine personnelle. Donc sur les machines de TP vous pouvez utiliser la commande suivante :

```
ocamlc -c -I +unix -I /opt/opam/default/lib/graphics CPgraphics.ml
```

et sur ma machine j'utilise :

```
ocamlc -c -I +unix -I ~/.opam/default/lib/graphics CPgraphics.ml
```

On peut bien entendu mixer le chargement de bibliothèques prédéfinies avec le chargement directe des bibliothèques que vous avez avez créé. Par exemple, si votre fichier sources `battleship.ml` utilise `CPgraphics` qui lui-même utilise `Unix` et `Graphics`, vous pouvez le compiler avec la commande :

```
ocamlc -c -I +unix -I /opt/opam/default/lib/graphics CPgraphics.cmo battleship.ml
```

S'il utilise en plus la bibliothèque `AP1array`, il faut ajouter cette dernière comme dépendance dans la ligne de commande.

Le principe est exactement le même pour l'édition de lien :

```
ocamlc -o battleship -I +unix -I /opt/opam/default/lib/graphics AP1array.cmo
unix.cma graphics.cma CPgraphics.cmo battleship.cmo
```

Le même principe s'applique pour générer la documentation :

```
ocamldoc -html -d doc -charset utf8 -I +unix -I /opt/opam/default/lib/graphics
-I . battleship.ml
```

5 Présentation générale

Le principe du jeu est connu (*cf.* figure 1). On dispose de 2 grilles : à gauche celle de l'adversaire (qui dans notre cas sera l'ordinateur), à droite notre grille (celle du joueur). Au départ, chacun place ses bateaux : ceux de l'ordinateur restent cachés, tandis que ceux du joueur s'affichent (en gris dans notre cas).

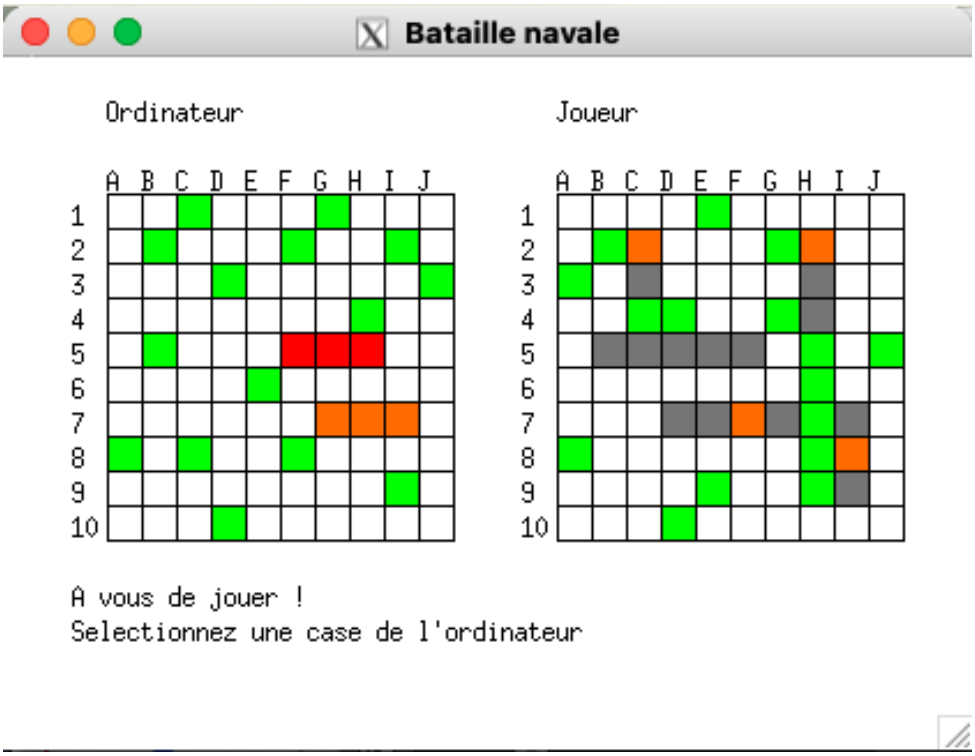

FIGURE 1 – Une image de la fenêtre graphique avec une partie en cours

Ensuite, le jeu proprement dit démarre. Le joueur et l'ordinateur sélectionnent alternativement des cases de le la grille de l'adversaire pour "tirer". Le tire peut tomber à l'eau (affiché en vert dans notre cas) ou sur un bateau (affiché en orange). Lorsque toutes les cases d'un bateau ont été touchées, le bateau coule (toutes ses cases sont alors affichées en rouge).

Lorsque tous les bateaux de l'adversaire ont été coulés, le joueur à gagné et le jeu s'arrête.

6 Itération 1 : affichage initial

Le but de cette première itération est de fournir un programme qui trace les grilles de départ dans la fenêtre graphique (voir la figure 2).

L'affichage initial dépend de différents paramètres. On peut citer :

— la largeur des marges `margin` (de 30 pixels matérialisé en rouge sur la figure),

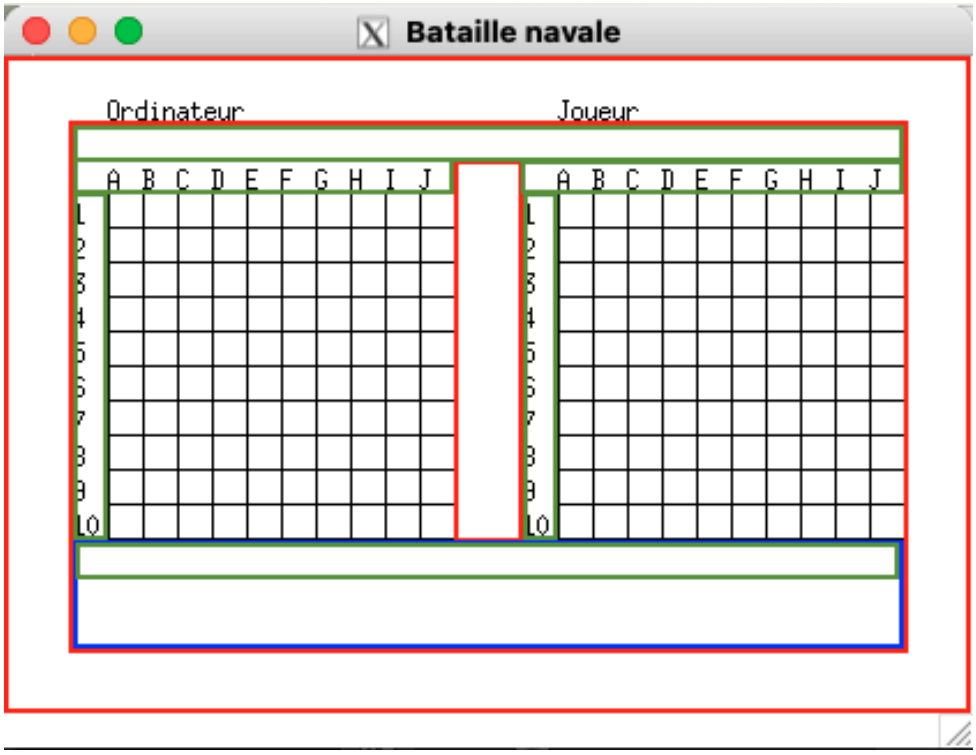

FIGURE 2 – Fenêtre graphique à l'ouverture du jeu avec en couleur les différentes zones

— la largeur des cases de la grille `cell_size`, qui est aussi la largeur de la zone d'écriture des numéros de ligne et des lettres de colonne et d'espace entre la grille et le texte (de 15 pixels matérialisé en vert sur la figure),
— la hauteur de la zone d'affichage des messages de consigne au joueur `message_size` (de 60 pixels matérialisé en bleu),
— la taille des grilles `grid_size` qui sont nécessairement identiques pour les deux grilles (de 10 cases, nous considèrerons uniquement des grilles carrées).

Définir le type structuré `t_params` qui comprend les paramètres cités ci-dessus et tous ceux que vous jugerez utile d'ajouter.

Définir une fonction `init_params` qui permet d'initialiser les paramètres du jeu avec un l'ensemble des valeurs d'une partie.

Il n'y a qu'une seule fenêtre graphique considérée par le programme OCaml. Il vous est donc conseillé de l'ouvrir une seule fois en début de séance. Vous pouvez utiliser la fonction `clear_graph` à chaque fois que vous souhaitez supprimer les affichages entre deux tests.

Définir une fonction `display_empty_grids` qui affiche les deux grilles vides, et le nom des deux joueurs ("Ordinateur" et "Joueur"), ainsi que ces sous-fonctions.

Définir la fonction principale `battleship_game` qui ouvre la fenêtre graphique aux dimensions appropriées, met à jour son titre et effectue les affichages adéquates. Puis appelez là.

Les tests unitaires et d'intégration des fonctions graphiques seront réalisés manuellement. Les tests fonctionnels sont à réaliser sur la version compilée.

7 Itération 2 : placement automatique des bateaux

Chaque bateau à placer est défini par son nom et sa longueur (le nombre de cases qu'il occupe). Modifiez le type `t_params` pour y ajouter un champs `ship_sizes` qui associe à chaque nom de bateau sa taille, sous la forme d'une liste de couples (nom, taille). Initialisez ce champ à :

```
[("Porte-avions", 5); ("Croiseur", 4);
```

`("Contre-torpilleur", 3); ("Contre-torpilleur", 3); ("Torpilleur", 2)]`

Définir le type matrice `t_grid` qui représente une grille (du joueur ou de l'ordinateur).

Le placement d'un bateau dans une grille est défini par la position de sa proue (l'avant du bateau) dans la grille et sa direction. Le placement automatique des bateaux de l'ordinateur se fait de la manière suivante : on tire aléatoirement une position et une direction, s'il est possible de placer le bateau à partir de cette position dans cette direction dans la grille, on le place, sinon on tire aléatoirement une nouvelle position et une nouvelle direction.

Une fois placé, un bateau est défini par son nom et la liste de ses positions. Définir un type structuré `t_ship` qui défini un bateau placé.

Écrire la fonction récursive `auto_placing_ships` qui place successivement tous les bateaux à placer dans la grille et retourne la liste des bateaux placés. Vous décomposerez cette fonction en environ 4 à 6 fonctions auxiliaires dont : la fonction récursive `positions_list` qui calcule la liste des positions d'un bateau à placer (à partir de sa position et sa direction de départ) et la fonction `can_place_ship` qui vérifie s'il est possible de placer un bateau dans la grille (à partir de la liste de ses positions).

Lorsqu'un bateau est placé dans la grille, il est affiché s'il s'agit de la grille du joueur, mais ne l'est pas s'il s'agit de la grille de l'ordinateur.

Écrire la fonction itérative `display_grid` qui colore les cases d'une des deux grilles dans la fenêtre d'affichage à partir de sa représentation matricielle. Pour cela vous coderez les fonctions `color_cell` qui colore l'une des case de l'une des grilles dans une couleur donnée et `cell_to_pixel` qui retourne la position du pixel en bas à gauche de la case donnée de l'une des deux grilles.

Étendre la fonction principale de l'itération 1 en plaçant aléatoirement les bateaux à placer du paramètre dans une `t_grille`, puis en affichant cette unique grille à la fois comme grille de l'ordinateur (les bateaux ne doivent pas apparaître) et comme grille du joueur (les bateaux doivent apparaitre).

8 Itération 3 : placement manuel des bateaux

Le joueur place manuellement ses bateaux en sélectionnant à la souris les cases de la grille où il souhaite les placer.

Nous allons utiliser l'espace d'affichage des messages (en bleu figure 2) pour lui indiquer le nom et la longueur du bateau à placer. Pour faciliter l'affichage des messages longs sur plusieurs lignes, les messages seront représentés par une liste de chaine de caractères. Ainsi chaque chaine de la liste sera affichée sur une ligne.

Écrire une fonction `display_message` qui efface le message précédent et affiche un nouveau message.

Écrire une fonction `read_mouse` qui attend le prochain clic de souris et retourne dans quelle grille on a cliqué (celle de l'ordinateur, celle du joueur, ou aucune des deux) et la cas échéant la position de la case dans laquelle on a cliqué.

Écrire une fonction récursive `manual_placing_ship_list` qui permet au joueur de placer l'ensemble des bateaux à placer dans sa grille et retourne la liste des bateaux placés. On pourra par exemple demander à l'utilisateur de cliquer successivement sur le première case du bateau, puis la deuxième. La fonction devra être ergonomique en colorant les cases sélectionnées (par exemple en jaune) et en permettant de recommencer la saisie en cas d'erreur (par exemple si la 2ème case n'est pas contiguë à la première ou si le placement du bateau est impossible à partir des cases sélectionnées. Pour ce faire vous décomposerez la fonction en plusieurs sous-fonctions

et réutiliserez les fonctions de l'itération précédente chaque fois que possible.

Définir un type structuré `t_battleship` qui contienne les 2 grilles de l'ordinateur et du joueur et la liste de leurs bateaux placés respectifs.

Écrire une fonction `init_battleship` qui a partir des paramètres du jeu place aléatoirement les bateaux sur la grille de l'ordinateur et permet à l'utilisateur de placer manuellement ses bateaux et retourne le `t_battleship` correspondant.

Modifiez la fonction principale pour réaliser toute la mise en place du jeu (le placement des bateaux dans les deux grilles).

9 Itération 4 : jeu du joueur

Lorsque le joueur tire, il y a plusieurs possibilités :

- s'il tire sur une case vide, le tire est manqué, la case doit s'afficher (par exemple en vert) pour que le joueur sache où il a déjà tiré,
- s'il tire sur une case de bateau, alors le bateau est touché, la case s'affiche en orange,
- s'il tire sur la dernière case du bateau (c'est-à-dire que toutes les autres cases du bateau sont déjà touchées), alors le bateau est coulé et toutes les cases du bateau s'affichent en rouge,
- s'il tire sur une case où il a déjà tiré, il ne se passe rien, les cases ne changent pas de couleur.

Il en sera exactement de même lorsque l'ordinateur jouera sur la grille du joueur.

Si les cases des tirs manqués, des bateaux touchés ou des bateaux coulés n'ont pas été prévues dans les cases des matrices de type `t_grid`, modifiez les pour prendre en compte tous ces cas. Modifier la fonction d'affichage d'une grille `display_grid` afin que tous les types de cases soient bien affichés dans une couleur différente.

Écrire la fonction `find_ship` qui recherche dans la liste des bateaux placés celui qui a été touché par un tir et la fonction `check_sunk_ship` qui vérifie si un bateau est coulé, c'est-à-dire si toutes ses positions sont touchées.

Écrire la fonction `player_shoot` qui permet au joueur de réaliser un tir. On pourra définir les fonctions auxiliaires `sink_ship` qui fait couler un bateau, c'est à dire met toutes ses positions à coulé dans la grille et `update_grid` qui modifie la grille suite au tir du joueur.

Complétez la fonction principale comme suit. Une fois le placement des bateaux réalisé (conformément à l'itération précédente), un nouveau `t_battleship` est défini avec la même grille (et les mêmes bateaux placés) pour le joueur et l'ordinateur. (Ainsi on pourra mieux tester les différents cas en sachant où sont les bateaux.) Puis demandez plusieurs tirs successifs du joueur, par exemple 5 tirs.

Pour permettre un affichage plus grand, modifiez les paramètres par défaut du jeu pour avoir une taille de cases de 20 pixels et des grilles de 15 cases. Si cette modification ne se limite pas à changer ces deux constantes, modifiez votre code pour faciliter les futurs modifications des paramètres.

Lors du test fonctionnel, prenez soin de tester les différents cas (manqué, touché et coulé).

10 Itération 5 : jeu complet

En réutilisant les fonctions de l'itération précédente, écrire une fonction `computer_shoot` qui permet à l'ordinateur de réaliser un tir. Si besoin vous généraliserez les fonctions précédentes pour permettre leur réutilisation. Dans cette itération l'ordinateur choisira aléatoirement la po-

sition de son tir, sans aucune stratégie.

Écrire une fonction `all_shoot` qui enchaine alternativement les tirs du joueur et de l'ordinateur jusqu'à ce que l'un des deux gagne.

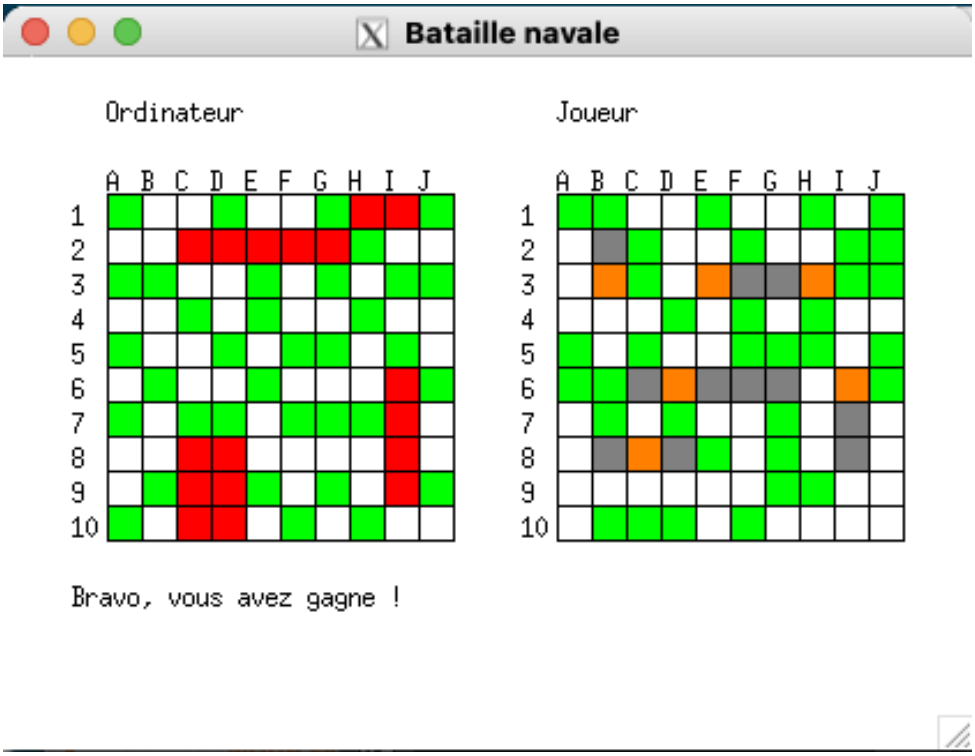

FIGURE 3 – Fenêtre graphique à la fin du jeu avec le message qui indique au joueur qu'il a gagné

Modifiez la fonction principale afin qu'elle enchaine un jeu complet (avec des grilles et placement de bateau différents pour le joueur et l'ordinateur), puis affiche un message qui indique au joueur s'il a gagné ou non (voir figure 3).

11 Itération 6 : extensions

Implantez une stratégie de jeu pour l'ordinateur qui améliore son jeu. Il pourra par exemple éviter de tirer sur des cases où il a déjà tiré, chercher à couler le bateau touché, etc.

Réalisez d'autres extensions convenues dans le groupe.

Toutes les extensions réalisées devront être soigneusement décrites avant leur conception afin que les besoins soient clairs pour tous.